Learning Programming with Erlang

or Learning Erlang with Ladybirds

Frank Huch

Christian-Albrechts-University Kiel
Institute of Computer Science
Olshausenstr. 40, 24098 Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract

This paper presents an interactive framework for pupils to learn the basic concepts of programming by means of the functional programming language Erlang. Beside the idea of the framework we also sketch the different learning targets and exercises to deepen programming skills.

The framework was successfully utilized in a programming course for pupils in their last three school years.

Categories and Subject Descriptors K.3.2 [*Computer and Information Science Education*]: Computer science education; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.6 [*Programming Environments*]: Integrated environments

General Terms Languages,Management

Keywords Erlang, Kara, Education, Programming

1. Introduction

In Germany the percentage of female students in computer science lays between 10 and 20%. To increase this percentage, our department decided to install a one week course for female pupils in the last three years at high school ("Oberstufe" of the "Gymnasium"), at the age of 15 to 18 years. The aim of this course is to give some insight of what computer science is and help the girls to detect whether this might be an interesting topic for them. Since computer science is not taught in every school in Germany and

girls often get the impression that people "hacking" computers are some kind of freaks who do not live in the real world, we decided to give this course as a kind of awareness training, what computer science really is and what one has to know/do when studying computer science in Kiel. The course was considered for one week and took place from Monday to Friday during the autumn holidays. We hade 20 female participants each time.

As a next step we had to fix the content of the course. It contains talks about the research of some of our groups, meetings with students and female graduates, a journey to a local IT company, and information about the curriculum. Furthermore, we think that programming is the key feature of computer science and, hence, almost half of the course was considered for teaching them programming. But what is the best way to teach girls at the age of 15 to 18 programming? An approach especially developed for girls of this age was the Kara system, from the University of Zurich, deployed in a similar course. The course in Zurich consists of three days learning programming with Kara and a final project, in which a hangman game is programmed in Java.

For our course we planed a similar schedule, but were not convinced of the contents from Zurich: We think their system is not the right choice to learn programming, as we sketch in Section 2. Furthermore, the project to implement hangman is no good choice. Much work has to be investigated into a graphical user interface, which is boring and is not appropriate to teach programming. Furthermore, it does not show aspects of computer science which go beyond programming, like designing a complex system in a team or developing system specifications.

As a solution, we developed a variant of Kara, in which students directly learn programming in Erlang [1, 2], which in our opinion has several advantages as we will sketch in Section 6. With this knowledge and Erlang's simple concept for concurrent and distributed programming, we were able to address a more interesting topic as the final project: we implemented a distributed chat in groups of 4 or 5 pupils. On the first day of the final project, we designed the system and specified a protocol, which was then implemented on the last day. The pupils were very impressed, what they were able to implement, and still use this chat on their own computers.

2. Kara from Zurich

There have been several toy like approaches to teach programming. One of the most famous are turtle graphics (conf. [9]), in which a picture is supposed to be painted by moving a turtle over the screen. The approach convinces by its attraction and fun to solve problems. However, this approach has the disadvantage that in turtle graphics you usually start with an empty screen and the exercise is given as a text. For beginners, this makes starting with turtle graphics a long way, since it takes time until the first interesting pictures are drawn. Furthermore, the turtle does not react on its environment. Hence, programs are less interesting and branching only considers values (e.g., counters of a loop). A program which has to react on its environment is a better motivation to learn programming. The pupils can directly relate such a program to the real world with cars, robots, and humans interacting.

As a solution more appealing approaches, like Karel the robot [10], were proposed in which an actor (e.g., a robot) has to solve tasks defined by its environment. A framework in the context of object oriented programming is the greenfoot environment, in which many similar environments have been defined [7]. As a special task

Erlang'07, October 5, 2007, Freiburg, Germany.

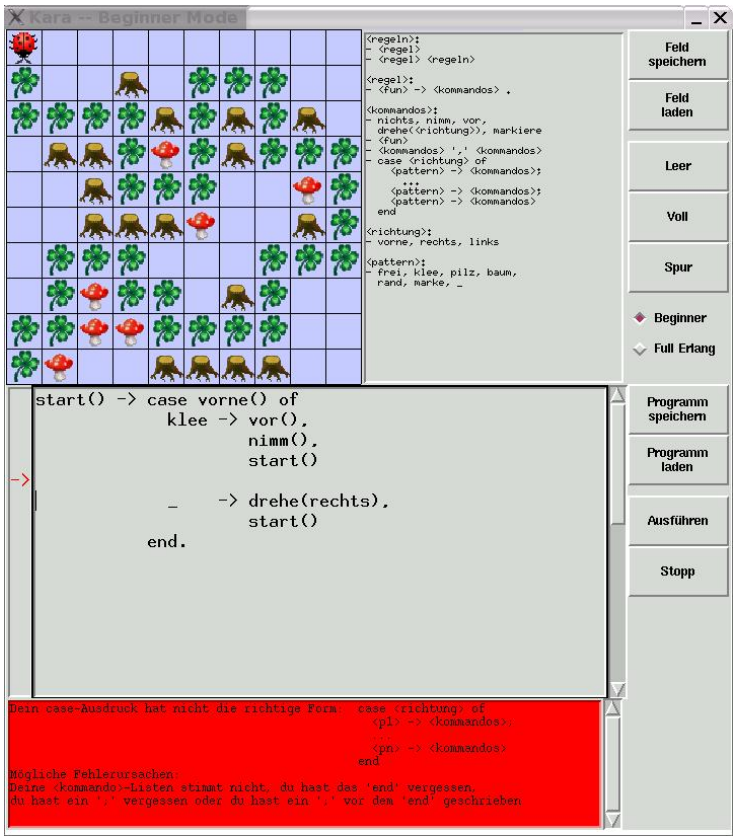


Figure 1. The Interactive Kara Development Environment

for girls the ladybird Kara[1] was invented [6]. Kara can walk around in her world, collect shamrocks, but should never walk onto a fly agaric, which kills her. Because of its attractiveness to girls, as reported from the group in Zurich, we decided to use Kara in our courses as well.

To control Kara, different environments have been defined [11]. To teach programming with Kara, the developers propose a two step approach: First, Kara is controlled by finite automaton to specify Karas behavior in dependence of its environment. Then in a second step the control is done by a programming language, usually Java (other imperative languages are available as well).

We do not agree, that this is a good way to teach programming. Automata are a means from theoretical computer science. They help students to learn state oriented, algorithmic thinking, but they do not provide abstraction mechanisms, which in our opinion are key features of writing good programs.

The only matter of abstraction provided by automata is the reuse of states, which is also possible in Erlang, where defined functions can be reused in the same way. Recursion (which in our experience can be easily understood by pupils) is not possible, which restricts to an sequential, operational view of algorithms. In contrast, defining functions like in Erlang and using them in arbitrary other positions provides a more abstract view of the system with thinking in a more denotational way. For special tasks, sub-computations can be defined (e.g. walk until you find a shamrock) in Erlang. They can be used in different situations, which leads to a better structure of the code. This is not possible using automata to control Kara.

Controlling Kara with Java can provide more abstraction mechanisms similar to the ones we teach them in Erlang (we do not introduce higher-order functions to the pupils), but with a lot more syntactic overhead and too many different concepts. For instance, the standard way of performing things several times is the use of a while or for loop in Java. Hence, the pupils first learn how to program in a purely sequential way without learning concepts of function abstraction with Java methods. A program consists of only one sequential method with several, maybe nested loops, but if an already programmed algorithm could be reused, the pupils have to use copy and paste, which results in unstructured, non-maintainable code. The abstraction of sub-computations is not possible.

Introducing Java methods, the pupils have to learn not only the concept of function abstraction. They also have to learn what classes and objects are, another non-trivial topic, for which in the context of Kara, it is difficult to find good applications. All objects of Karas world are already defined. For beginners, many concepts of Java will only be typed into the computer as non-understood syntactic overhead.

Finally, the approach taken in Zurich has a clear break when switching from Automaton Kara to Java. But concepts are different: instead of changing states in the automata, they have to use loops in Java. Hence they first learn how to program with goto and then they change to the more structured language Java not providing gotos. The drawbacks of programming with gotos were already discussed in 1968 by Dijkstra [4]. So why should the students learn a concept which is known to have downsides.

A similar consensus is taken in [3], which extracts faults of general-purpose languages for teaching programming to beginners:

[1] Since the course is given for girls, Kara is a female ladybird.

- They "... are too big and too idiosyncratic. ... Instead of emphasizing fundamental principles, the languages evoke secondary notions which reflect the subtleties of the given language and its implementation."
- They "... provide little leverage for understanding their basic actions and control structures. The languages are not visual and their basic functions are carried out behind an opaque barrier."
- They "... are oriented on number and symbol processing". Hence, "... the first possible problems used in teaching the language are far from the students' everyday experience and are not attractive for them."

Furthermore, they extract, that these drawbacks become even more relevant, the younger the taught people are, which we have to consider when teaching pupils.

We think that Kara elegantly solve the last two points, since it is a nice environment, defining interesting tasks and elegantly visualizing the effects of a program execution. However, the first point is not sufficiently solved by the approach taken in Zurich and remains a problem programming Kara with common imperative general-purpose languages, like Java.

As a solution [3] proposes the use of mini-languages to control toy environments like "Karel the Robot". However, there remains one big disadvantage of using mini-languages: finally, there is a point where programming has to go beyond the toy environment and real applications have to be developed. Here you usually have a big gap between the mini-language and general-purpose languages, such that it may become hard to transfer the learned concepts to the new language.

We think that Erlang (in a restricted fashion) is at least as good as proposed mini-languages. Furthermore, Erlang is also a general-purpose language and can also elegantly be used for concurrent and distributed programming. Although these are advanced topics, the pupils know their applications from using the Internet, especially, when developing a standard application, like a chat. Hence, the final project also is a real world application, in contrast to programming a hangman game. After having implemented the chat, they understand the principles of distributed applications.

3. Erlang and Kara

For absolute beginners it is very important to have an integrated environment which elegantly supports programming. A screenshot of the system is presented in Figure 1[2]. It provides the following facilities:

- Karas world – the field in which Kara lives. It also contains trees, fly agarics and shamrocks. Fields can be modified with the mouse, saved to and loaded from a file.
- An editor for the program text and buttons to start and stop the program execution.
- An EBNF describing Karas initial programming language.
- An error field, for the presentation of compiler and runtime errors. In addition to the error message, an arrow points next to the program editor points to the program line, in which the error occurred. The error messages are improved, such that they are as comprehensible for beginners as possible, in contrast to standard Erlang error messages.

[2] The language of the system is German, since it was used to teach Germans. In the remaining paper, we use English translations, especially in the programs.

$$
\begin{array}{lcl}
Prg & ::= & RulePrg \\
 & | & Rule \\
Rule & ::= & Func\texttt{() ->}\ Cmds\ \texttt{.} \\
Cmds & ::= & Cmd\ \texttt{,}\ Cmds \\
 & | & Cmd \\
Cmd & ::= & \texttt{go()} \mid \texttt{take()} \mid \texttt{mark()} \mid \texttt{nothing()} \\
 & | & \texttt{turn(}Dir\texttt{)} \\
 & | & \texttt{case}\ Dir\texttt{()}\ \texttt{of} \\
 & & \quad Pat_1\ \texttt{->}\ e_1\texttt{;} \\
 & & \quad \ldots \\
 & & \quad Pat_n\ \texttt{->}\ e_n \\
 & & \texttt{end} \\
Dir & ::= & \texttt{left} \mid \texttt{rigth} \mid \texttt{front} \\
Pat & ::= & \texttt{free} \mid \texttt{shamrock} \mid \texttt{agaric} \mid \texttt{tree} \mid \texttt{border} \\
 & | & \texttt{\_} \\
Func & ::= & [\texttt{a}-\texttt{z}]\ [\texttt{a}-\texttt{z}, \texttt{A}-\texttt{Z}, \texttt{0}-\texttt{9}]^* \\
\end{array}
$$

Figure 2. Syntax of the initial language

3.1 Programming Kara

Instead of programming boring algorithms on numbers, like computing the greatest common devisor of two numbers, the pupils directly have to write programs to control Kara in its world. For this purpose, the system provides special Erlang commands to control Kara:

- `go()` – move Kara one field ahead
- `turn(`*direction*`)`, where *direction*$\in$`{left,right}` – to turn Kara
- `take()` – pick up a shamrock at the actual position, if possible.
- `mark()` – put a mark (a turned shamrock) on Karas actual position
- `nothing()` – perform no action. Sometimes useful when the syntax requires some code, but nothing should be done.

Furthermore, Kara can detect its environment by means of direction functions

```
left(), right(), front()
```

which return the values

```
free, shamrock, agaric, tree, border
```

in dependence of the field position in the checked direction. In the initial language, the pupils are supposed to use the direction functions only in the context of a `case` expressions.

4. Teaching Levels

Similar to the DrScheme system [5], we define different levels of the language, in which the pupils are supposed to learn different aspects of programming. However, these levels are tailored differently. The first level in our framework can be much simpler than in DrScheme, since the Kara environment allows interesting tasks which can be solved even without using parameters. This is not possible in the general purpose language approach of DrScheme. The former levels slightly differ for the differences between Erlang and Scheme (e.g. Erlang does not provide nested function definitions).

4.1 The Initial Language

We start with a very restricted version of Erlang without parameters, variables and data structures. The pupils can use sequential composition and function abstraction to define complex command sequences. Branching in the program code, in dependence

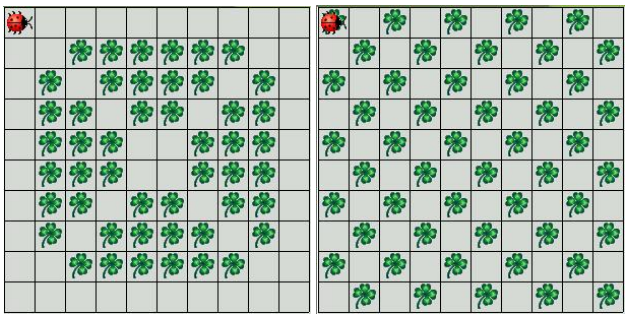


Figure 3. Patterns to create

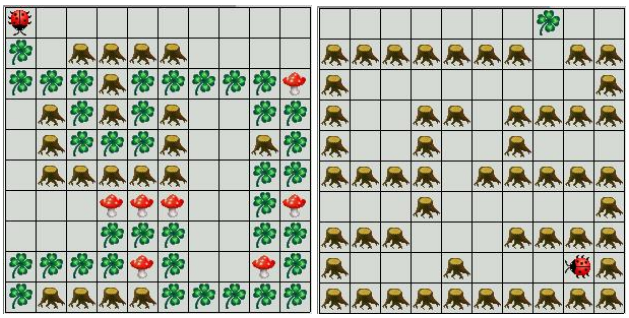


Figure 4. Trail and Labyrinth

of Karas environment, is possible by means of a `case` expression, which may only contain the restricted patterns `free, shamrock, agaric, tree, border` and the wildcard _, which matches arbitrary values.

The EBNF of the initial language is presented in Figure 2. Note, however, that this is a valid subset of Erlang which can later be extended such that the pupils can stepwise learn full Erlang. The execution always starts with the null-ary function `start()`. Further functions of arity zero can be defined. Initially, we call functions *named sequences* and explain their semantics by a simple inlining-example:

```
start() -> takeThree(), turn(left), takeThree().

takeThree() -> goTake(), goTake(), goTake().

goTake() -> go(), take().
```

which has the same semantic as the program

```
start() -> go(), take(), go(), take(),
           go(), take(),
           turn(left),
           go(), take(), go(), take(),
           go(), take().
```

Hence, as one of the first programming principles, we introduce function abstraction, which is willingly adopted by the pupils in the exercises. The step to the (tail-)recursive case is then straight forward to the pupils and demonstrated by simple examples in a rewriting semantics, e.g.:

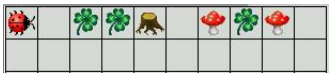
```
start() -> case front() of
             agaric -> nothing();
             free   -> go(), start()
           end.
```

which makes Kara move over free fields until she detects a fly agaric in front.

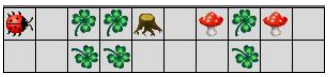
The start-up with this small language only takes 45 minutes, and after this small lecture the pupils directly start programming several exercises. Since they get into contact with compiler and run-time errors for the first time, we provide intensive help by supervisors in this phase. For instance, it is very difficult for them to understand that there is no semicolon allowed after the last case alternative.

The exercises start with picking up some shamrocks in a fully filled field, such that the patterns of Figure 3 can be created. When programming these exercises, we set a high value on implementing the algorithm as flexible as possible, which means that the program should not only work for the given context, but also for other fields of even other sizes. Furthermore, we set a high value on a good layout (the first programs often only contain one line of 200 characters length) and the reuse of already defined functions.

As a next step the pupils have to copy all shamrocks in one line. Applying this program the situation



should result in a situation like the following, where the final position of Kara is not important:



The interesting task in this exercise is that Kara has to detect different items in her world and react accordingly. Especially she has to finish copying at the border. We obtain different solutions for this exercise. In the nicest solution Kara walks underneath the line to be copied and alternately looks to the left (to detect shamrocks) and to the front (to detect the border).

In the next exercises, they have to collect a trail of shamrocks and escape from a labyrinth, conf. Figure 4.

To demonstrate, how elegantly these exercises can be solved within this initial language, we stepwise present the code usually written by the pupils to collect a trail of shamrocks. They usually start with the following program:

```
start() -> case front() of
             shamrock -> go(), take(), start();
             _        -> turn(left), start()
           end.
```

This program already collects a full trail of shamrocks. The only fault of this solution is that the program does not terminate. At the end of the trail Kara turns infinitely to the left. As an improvement, we ask the pupils to collect the shamrocks more goal-oriented, i.e. only turn if it is valid to turn. Usually, this directly results into a perfect solution:

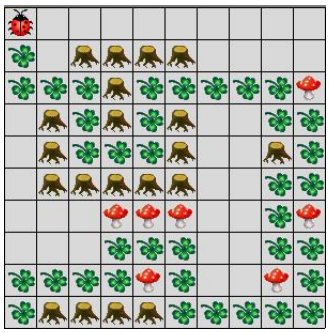
```
start() ->
  case front() of
    shamrock -> go(), take(),
                start();
    _        ->
      case left() of
        shamrock -> turn(left),
                    start();
        _        ->
          case right of
            shamrock -> turn(right),
                        start();
            _        -> nothing()
          end
      end
  end.
```

4.2 Recursion

At the next level, we generalize the use of recursive calls, by asking the pupils, what the following program does, if Kara stands in front of three shamrocks:

```
start() -> case front() of
             shamrock -> go(), take(),
                         start(),
                         mark(), go();
             free     -> go()
           end
```

Some pupils immediately know the correct answer. Some others have to think about it for a while, and get the crucial point while discussing it. The rest of the pupils understand this program after executing the program with a rewriting semantics. This lecture takes only 15 minutes and they can generalize the concept of recursion with the next two exercises: As a first step they have to modify their program to collect a trail shamrocks (cf. Figure 4) such that after collecting all shamrocks Kara runs back to the initial place and marks every collected shamrock on this way back, which results in



where the shamrocks are replaced by marks and Kara is turned by 180 degree in contrast to the initial situation in Figure 4.

Starting from the solution for collecting a trail of shamrocks, marking the trail after reaching its end is quite simple. Instead of doing nothing at the end of the trail Kara has to turn around and, on the way back, she has to reverse every action:

```
start() ->
  case front() of
    shamrock -> go(), take(), start(),
                mark(), go();
    _        ->
      case left() of
        shamrock -> turn(left), start(),
                    turn(right);
        _        ->
          case right of
            shamrock -> turn(right), start(),
                        turn(left);
            _        -> turn(left), turn(left)
          end
      end
end.
```

As a next and final step for the first day, the pupils are expected to collect arbitrarily branching trails (or nets) of shamrocks, as shown in Figure 5. They have to implement backtracking. Starting from the last program the step to an implementation of backtracking is not far. Instead of marking the places when walking back, they can look for alternative branches still containing shamrocks. In the `start()` function of the previous they can simply replace

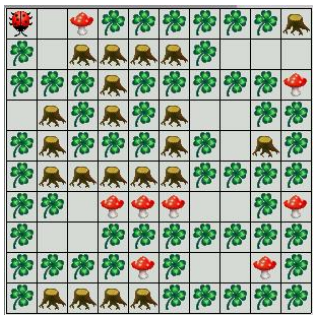
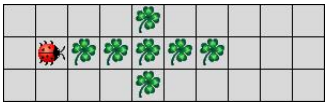


Figure 5. Branching Shamrock Trail

the call `mark()` by a call to a new function `checkBranch()`, which can be defined as follows:

```
checkBranch() ->
  case left() of
    shamrock -> turn(left), start(),
                turn(left)
    _        -> nothing
  end,
  case right() of
    shamrock -> turn(right), start(),
                turn(right)
    _        -> nothing
  end.
```

A common mistake the pupils make when defining this function is that the case expression for detecting shamrocks on the right is nested into the case for detecting no shamrock on the left, similar to the definition of the function `start()`. We demonstrate this mistake to the pupils by the following situation



where Kara does not collect the upper shamrock. Analyzing the mistake, deepens their insight into recursion a lot.

In the context of this exercise, we also briefly explain the notion of an invariant of the algorithm. The invariant of the function `start()` is that if Kara is located at a position p before a call of `start()`, then after the execution of `start()` Kara is located at the same position p but looking into the other direction. With this invariant in mind, it is clear in which direction Kara has to turn in the definition of `checkBranch()` after collecting a branching trail of shamrocks.

4.3 Parameters

On the second day we motivate the use of parameters by counting shamrocks and presenting the result to the user by putting the same amount of marks. In this context we introduce the notion of variables, parameters, and present simple predefined operations, like `+`, `-`, `==`. To avoid pupils from thinking too imperative, we use the notion of knowledge of a function, e.g., to count something. We avoid to say that variables are updated, but call a function with another knowledge. This is often more problematic for pupils who already learned imperative programming in school, than for absolute beginners.

Furthermore, we introduce the notion of atoms in Erlang, generalize pattern matching to arbitrary atoms (including numbers) and introduce the binding of variables by `=`. Surprisingly, the concept of parameters and variables is more problematic for the pupils, than

recursion[3]. As a consequence, we investigate a lot of time in practicing programming with parameters.

The exercises in this section are checking that two lines in Karas field contain the same amount of shamrocks, counting all shamrocks in the whole field and putting the same amount of marks in the end and finding that line on the field in which the maximum number of shamrocks is located. As an additional exercise for the fast pupils, some of these exercises can be solved by using recursion instead of parameters.

When implementing these programs we point out, how important it is to develop reusable software. A good example is the function to count shamrocks within a line:

```
count() -> case front() of
             shamrock -> go(), count()+1;
             free     -> go(), count();
             border   -> 0
           end.
```

or in a tail-recursive version:

```
count(N) -> case front() of
              shamrock -> go(), count(N+1);
              free     -> go(), count(N);
              border   -> N
            end.
```

This function can be reused in each of the exercises. For instance, in the first exercise it can be used twice to count the number of shamrocks in each of the two lines and in the last exercise it can be used to count the number in each line of the field.

4.4 Data Structures

As a next step we introduce data structures. We start with tuples to combine arbitrary values, motivated by an example, in which we count shamrocks, trees and agarics while running along a line and looking to the right.

```
count(Shams,Trees,Agarics) ->
  case front() of
    border -> {Shams,Trees,Agarics};
    free ->
      case right() of
        shamrock -> count(Shams+1,Trees,Agarics);
        tree     -> count(Shams,Trees+1,Agarics);
        agaric   -> count(Shams,Trees,Agarics+1)
      end
  end.
```

As a dynamic data structure we introduce Erlang lists and present basic functions, like `append/2`(`++`), `length/1`, and `reverse/1`. As an example, we present a function which records the items of a line within a list:

```
collect() -> case front() of
               border -> [];
               free   -> Item = right(),
                         Line = collect(),
                         [Item|Line]
             end.
```

Here the evaluation order and the side effects to Karas world become crucial, which we discuss by changing the order in which the actual item is collected (`Item = right()`) and the recursive call (`Line = collect()`) is performed. We avoid ambiguities resulting from the evaluation order of subexpressions, like defining the case branch for `free` as `[right()|collect()]` by sequentializing crucial calls by comma, Erlang's sequential operator.

To practice programming with data structures the pupils have to write a program which compares whether two lines are similar or the mirror image of each other. As a second exercise, the pupils have to remember one line and search the remaining lines for an identical one. Here we obtain many different solution. The clever students once define a function to get a list representing the items of one line. This function is then reused for every line of the field and the resulting lists are then compared. This solution is very elegant from the software technical point of view (which in our opinion is the most important thing when learning programming). However, a more efficient solution is possible which compares the initially read line, successively with every line. In this solution Kara can already stop comparing the lines, when an initial part is different. Hence, she does not have to walk through every line completely.

4.5 Programming with Erlang beyond Kara

After three days programming with Kara, we introduce Erlang's features for concurrent and distributed programming to the pupils, which usually results in an "Oh, what a petty!", when telling them that we do not program with Kara anymore.

We give a brief introduction to concurrent and distributed programming in Erlang. The notions of "processes", "pids", and "message passing" are easily introduces with a comparison to the real world, in which also multiple people live, can be distinguished by their addresses, and communicate with each other. When explaining Erlang's communication concept it becomes very handy that receive works similar to Erlang's case expression which we already introduced in the first lecture and used for branching in every Kara program.

After this last lecture we split the course into groups of 4 or 5 pupils. Then each of these groups develops a protocol for a chat, consisting of a server, a client, and a keyboard process. The groups are guided by a supervisor and the specifications are prepared on whiteboards, similar to the specifications presented in [8]. Developing the protocols in a group on a blackboard is a new experience for the pupils, which we consciously integrate into the course, to demonstrate than computer science is more than hacking. Soft skills, like the ability to communicate, and the development of an accurate specification are at least important for a successful project as being a good programmer.

5. Implementation of the Kara System

To provide good error messages for the pupils, we implemented a simple interpreter for the initial language. Error messages are optimized with respect to this restricted language and the relevant parts of the EBNF are presented to the pupils. Furthermore, we give specialized hints to common errors, like forgetting a semicolon inbetween the different case branches or writing a semicolon after the last case branch. Beside printing the error message, an arrow beside the program editor points to the error line. Runtime errors are presented in a similar way, e.g., a pattern matching failure or the case when Kara runs onto a fly agaric.

For programming with parameters and data structures, we do not provide an own interpreter. Instead we use the Erlang system. However, Kara is supposed to teach programming to the pupils. We do not want to confuse them with module heads, import- and export lists. Hence, we add a static header file to the pupils program, which provides a module header, an import list for the allowed Kara commands, and an export list containing the function `start/0`. The resulting module is compiled by the Erlang runtime system and then started as a separate Erlang process which communicates with the control process of Karas world.

[3] In contrast to the students assisting in this course, who have a lot more problems in understanding recursion and backtracking, than programming with parameters.

The error messages of the compiler, as well as the runtime error messages of the Kara program are analyzed and presented in a beginner conform way to the user. Again the line in which the error occurred is marked by an arrow beside the program editor.

6. Advantages of Erlang-Kara

As already discussed in Section 2, we think that controlling Kara with finite automaton as well as programming Kara with Java is no appropriate approach for learning programming. Using Erlang, we restrict to a very pure programming language without syntactic sugar. No syntactic overhead is necessary, not even the module headers usually necessary in Erlang programs. Beginners can concentrate on the real challenges of programming. They do not have to fight different concepts for similar things, like different kinds of loops. The only concept to perform things several times, is recursion, which has a intuitive, simple semantics. Most students are able to implement backtracking on their first day!

The concept of reacting on the world by means of pattern matching in a case expression is very intuitive, in contrast to complicated if-then-else cascades necessary in Java. As a side effect, the pupils are perfectly prepared for pattern matching on data structures or in the context of the `receive` expression, which work similar in Erlang.

Another opportunity of our approach is that here is no change in the language. We directly start with (a restricted version of) Erlang, which is step by step extended with new features, like parameters and data structures. Programming in a functional language is a much more abstract concept. It results in a more abstract understanding of problems by the pupils and better solutions for complex problems.

A final good point of using Erlang is that some pupils already learned Java in School. Using a completely different language and concept relativizes their advances. Sometimes (e.g. when they want to modify variables) the absolute programming beginners are even a bit ahead.

For future work we want to improve the user interface, e.g., with a configurable size of the environment and make the execution speed scalable by the user. Furthermore, we plan to integrate Erlang's concurrency features (which are now taught independently from Kara) into Kara, which would for instance be useful if multiple ladybirds have to solve a problem cooperatively.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[2] Joe L. Armstrong. The development of erlang. In *International Conference on Functional Programming*, pages 196–203, 1997.

[3] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. Mini-languages: a way to learn programming principles. *Education and Information Technologies*, 2(1):65–83, 1998.

[4] Edsger W. Dijkstra. *Go to statement considered harmful*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA, USA, 2001.

[6] W. Hartmann, J. Nievergelt, and R. Reichert. Kara, finite state machines, and the case for programming as part of general education. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Poul Henriksen and Michael Kolling. Greenfoot: Combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 73–82, Vancouver, BC, CANADA, November 2004. ACM.

[8] Frank Huch. Erlang specification method - a tool for the graphical specification of distributed systems. In *Seventh International Erlang/OTP User Conference*, 2001.

[9] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[10] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1981.

[11] Raimond Reichert. Programmieren lernen mit kara, available from `http://www.swisseduc.ch/informatik/karatojava/`.